1992

# InterBase: An Execution Environment for Global Applications over Distributed, Autonomous and, Heterogeneous Software Systems

Ahmed K. Elmagarmid
*Purdue University*, ake@cs.purdue.edu

Jiansan Chen

Weimin Du

Omran Bukhres

Rob Pezzoli

Report Number:
92-016

# INTERBASE: AN EXECUTION ENVIRONMENT FOR GLOBAL APPLICATIONS OVER DISTRIBUTED, AUTONOMOUS, AND HETEROGENEOUS SOFTWARE SYSTEMS

Ahmed K. Elmagarmid
Jiansan Chen
Weimin Du
Omran Bukhres
Rob Pezzoli

CSD-TR-92-016
March 1992

# InterBase: An Execution Environment for Global Applications over Distributed, Autonomous, and Heterogeneous Software Systems

Ahmed K. Elmagarmid and Jiansan Chen
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{ake, jchen}@cs.purdue.edu

Weimin Du
Database Technology Department
HP Laboratories Division
Palo Alto, CA 94304
du@hpl.hp.com

Omran Bukhres
C. S. and I. S. Department
Moorhead State University
Moorhead, MN 56563
bukhres@mhd1.moorhead.msus.edu

Rob Pezzoli
BNR Inc.
35 Davis Drive
RTP, NC 27709-3478
ROBP@BNR.CA

**Abstract**

Existing and well entrenched hardware and software systems are the product of lengthy and individual developmental histories. The introduction of harmonious cooperation among such systems carries the potential for great increases in productivity and improvement in applications processing. However, the utilization of these heterogeneous components is hampered by the absence of an integrated system that would allow the development of global applications requiring communication and cooperation among existing systems. Designed to integrate pre-existing systems over a distributed, autonomous, and heterogeneous environment, and to support global applications while retaining local autonomy, the InterBase System overcomes this heterogeneity barrier. The fundamental underpinning of this system is the InterBase Parallel Language (IPL), which provides an interface allowing users to write global applications over such an environment. IPL supports flex transactions [ELLR90] and provides commitment constructs that allow users to define their own commitment protocols; both features are superimposed upon the component pre-existing systems. Remote System Interfaces are provided as a mediator to deal with the heterogeneity of these pre-existing systems and to present a uniform system-level interface to IPL programs and their interpreter. A Distributed Flexible Transaction Manager serves to interpret IPL Programs and to coordinate their executions. Based on the concept of Quasi Serializability [DE89, ED91], a Distributed Concurrency Controller has been developed to manage the parallel read/write accesses to the pre-existing systems. The system presented here is modular and is ideally suited to heterogeneous hardware, software, and network environments, particularly multidatabases. This paper explores the issues of system architecture, language design, concurrency control, and system interfaces in such a project. As a case study, we also discuss such a system which is currently in use at BNR Inc.

# Contents

# 1 Introduction

Heterogeneous hardware and software systems typically arise in the process of fulfilling diverse computational and information processing requirements. The overall heterogeneity of the computing environment increases as each new generation of a system is put into operation. This heterogeneity becomes a shaping constraint in the development of global applications, which access data and request services from several different systems. A multidatabase is such a heterogeneous system. A multidatabase integrates pre-existing and independent database systems in such a way as to support global applications accessing more than one element database. A multidatabase interoperation provides an integrated view of the data and the resources of these applications, with no violation of the local autonomy of the element databases or the autonomy of their administration.

Consider, for example, a travel agent information system. To plan a trip, a client needs to book airline tickets, rent cars, and reserve hotel rooms. Independent systems have been developed for each of the services. For example, each airline company has its own system for flight ticket reservation. Users can make reservations by accessing these independent systems. Such an approach, however, is not only inconvenient and inefficient, but also unable to support global applications requiring different services, such as locating the most direct combination of flights (possibly involving more than one airline) from A to B. To support global applications of this kind without sacrificing local applications, it is necessary to integrate the pre-existing systems to provide users a uniform view of and a consistent access to the element systems. Element systems should be integrated not only at the logical level (to provide global schemas and views), but also at the system level (to support global transactions). For example, a user who requests a flight ticket and a hotel room would like either both or neither of the requests to go through. It is also important to coordinate concurrent requests so that no ticket is sold more than once.

The accomplishment of system level integration, however, is difficult in multidatabases, due to the autonomy of the element databases. Local autonomy in a multidatabase is an artifact of the history of its element databases, which were originally independently developed and administrated. The retention of local autonomy facilitates flexible integration of element systems and guarantees that old applications continue to be executable without modification. For example, in the above travel agent information system, it is important that each element system continue to process local applications as it had prior to integration and to maintain full control over its resources. It may therefore be impossible to ensure the traditional ACID (automicity, consistency, isolation, and durability) properties of transactions, because a local database system, being autonomous, can unilaterally abort a global transaction. Consider, for instance, a user who wishes to purchase a ticket from New York to Chicago on United Airlines and another ticket from Chicago to Lafayette, IN on American Eagle. Since the two requests are independently processed by separate systems, one of them may be rejected while the other is approved. Similarly, it is difficult to coordinate the execution of global applications while they are independently scheduled by local systems. Therefore, more flexible transaction models and transaction management strategies are both useful and necessary in supporting system level integration.

In this paper, we present a system solution to the above problems, developed in the course of

investigations in the InterBase project at Purdue University. The purpose of this project is the design and implementation of a global environment for applications accessing (i.e., reading/writing) over multiple systems. A prototype, called the InterBase System, is designed to integrate pre-existing systems over a distributed, autonomous, and heterogeneous environment. The system allows users to write global applications over heterogeneous systems by employing a uniform language called the InterBase Parallel Language (IPL). IPL supports flex transactions [ELLR90] and provides commitment constructs that allow users to define their own commitment protocols, both of which are superimposed on those of pre-existing systems. Our prototype uses specially designed agents, Remote System Interfaces, to deal with the heterogeneity of these pre-existing systems and to provide a uniform system level interface to IPL programs and their interpreter. A Distributed Flexible Transaction Manager is provided for the InterBase system; it interprets IPL Programs and coordinates their executions. A Distributed Concurrency Controller is also developed, which maintains quasi serializability [DE89, ED91], a weaker consistency criterion than the traditional serializability, without violating local autonomy. Our solution is modular and can be implemented in heterogeneous hardware, software, and network environments such as multidatabases.

The problem of integrating pre-existing database systems has been studied by many other researchers, and several other prototypes have also been developed. Examples include Multibase [LR82], Mermaid [TLW87], MRDSM [Lit85], ADDS [BOT86], and DATAPLEX [Chu90]. Most of these approaches, however, are either a front-end to multiple databases or support only logical level integration. Some, such as ADDS, do support system level integration. These, however, are based on the traditional transaction model and transaction management strategy and impose significant restrictions on element databases. The InterBase system is new in that it emphasizes system level integration. It not only guarantees correct synchronization among operations issued by concurrent applications, but also supports an enhanced flex transaction model and a commitment protocol. Additionally, in designing the InterBase System, we make no assumptions about the nature of pre-existing systems. Global applications in the InterBase System can therefore incorporate not only database systems but also non-database systems over a wide network of mainframes, workstations, and mini-computers. The InterBase system is also designed to integrate distributed, heterogeneous, and autonomous applications on the basis of the flex transaction model [ELLR90, LEB92], thus overcoming the limitations of the traditional transaction model.

The body of this paper is organized as follows. Section 2 provides an overview of the InterBase System, while Section 3 discusses the IPL Language. The Remote System Interfaces are addressed in Section 4. The Distributed Flexible Transaction Manager and a Semantics-based Commitment Protocol are set forth in Section 5 and the Distributed Concurrency Control in Section 6. A case study of the InterBase System at BNR is given in Section 7. Finally, Section 8 outlines directions for future work and presents conclusions gleaned from our current investigations.

Throughout this paper, a *global transaction* refers to a global application, and a *subtransaction* refers to a subtask of a global application which is executed on a Local Software System (LSS).

# 2 Overview of the InterBase System

This section delineates the architecture of the InterBase System. We shall describe the various components and modules of the system and briefly elucidate their mutual interactions. The InterBase System is designed to allow users to write global applications over a distributed, autonomous, and heterogeneous computing environment, such as a multidatabase environment, while retaining the autonomy of LSSs. By making no assumptions regarding LSSs and by using the standard LSS interfaces only, the InterBase System succeeds in preserving this autonomy.

A version of the InterBase System currently in use at Bell Northern Research Inc. will be discussed extensively in Section 7 [PE91].

## 2.1 The Architecture of the InterBase System

The major components and modules of the InterBase System and the relationships among them are presented in Figure 1. Arrowed lines indicate the flow of commands and data among the modules of different systems, while unarrowed single lines represent such flow between Remote System Interfaces (RSIs) and LSSs. The unarrowed double line indicates that subtransaction schedulers are portions of RSIs. At present, the InterBase System runs on an interconnected network with a variety of hosts that include Sun, HP and NeXT workstations, Sequent machines, IBM mainframes, and IBM/PCs.

The architecture of the InterBase System is designed in such a way that all its major modules, except the LSSs, interface with the Distributed Flexible Transaction Manager (DFTM). DFTM interprets and coordinates the execution of global transactions, which are in the format of IPL, over the entire system. RSIs ensure a uniform interface to DFTM and deal with the heterogeneity of the LSSs. A user can invoke high level user interfaces, such as a graphical interface, to make a query to the InterBase System; a User Interface will translate the query into an IPL text, and the text will then be sent to DFTM for execution. A user with a good grasp of LSSs and a fluency in IPL can also write and send IPL texts to DFTM for direct execution. An IPL text from either source is executed by DFTM as a global transaction over the InterBase System. Assisting in this process is the Distributed Concurrency Controller (DCC), consisting of a Group Manager and Subtransaction Schedulers, each of which is a portion of an RSI. DCC is so named because it is based on distributed algorithms, these will be discussed in Section 6. DCC is used to manage the parallel access of global transactions over the InterBase System.

In the following subsections, we will provide a more detailed explanation of these modules and of their interrelationships.

## 2.2 InterBase Parallel Language

Global applications in the InterBase System are currently written in the InterBase Parallel Language (IPL), which has been designed to support a flex transaction model [ELLR90]. In this language, subtransactions of global transactions are executed in parallel whenever possible. While retaining

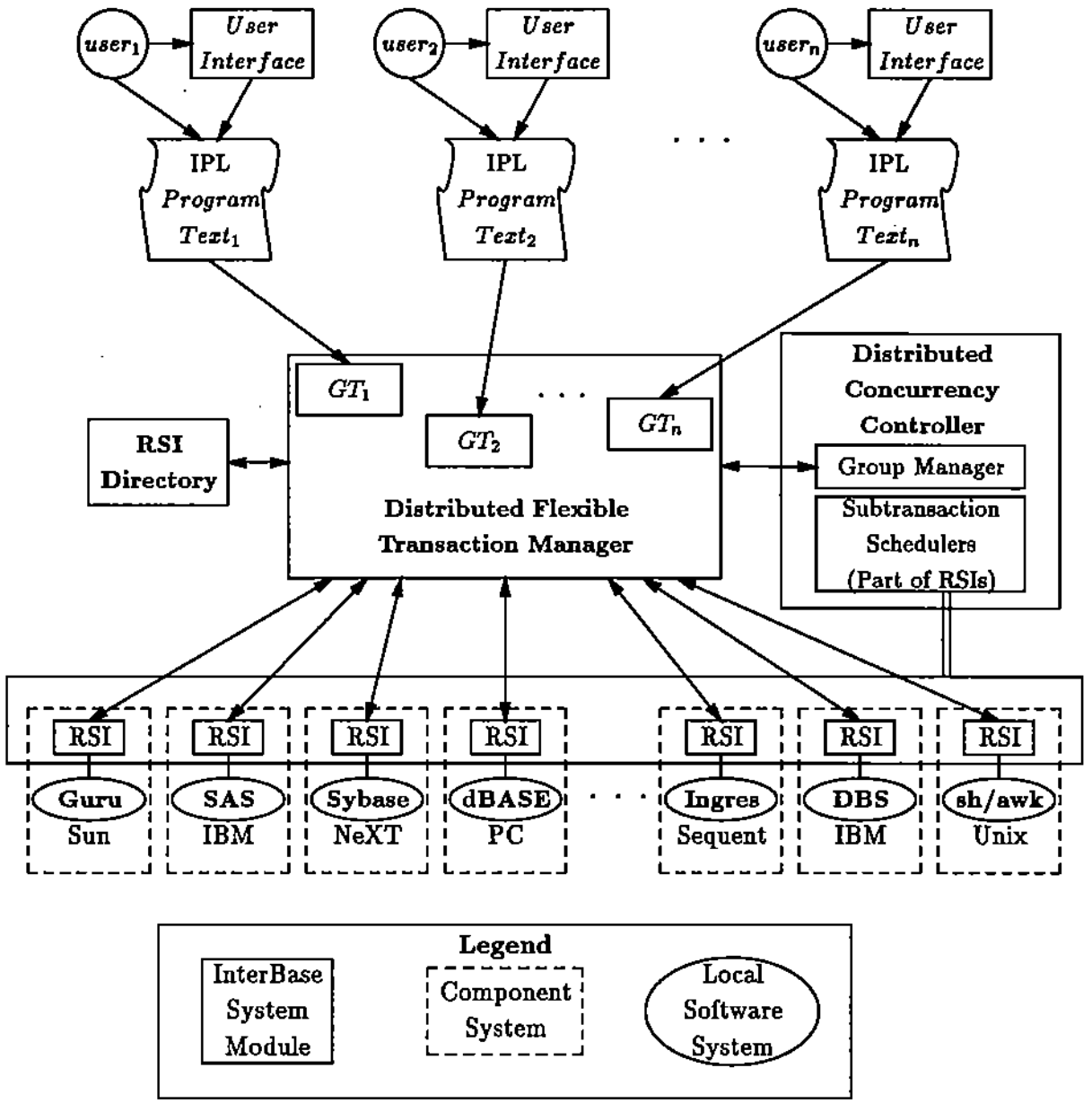


Figure 1: The architecture of the InterBase System

local autonomy, IPL allows users to specify all actions associated with a global transaction, such as their sequence and logical dependencies among subtransactions. A detailed discussion of IPL is provided in Section 3. Its syntax and semantics appear in the Appendix A and Appendix B, respectively.

### 2.3 RSIs and RSI Directory

A Remote System Interface (RSI) is an intermediary located between DFTM and an LSS. An RSI translates a command text from DFTM into a format understandable by the LSS and sends it to the LSS for execution; it also manages data flow and data format transformation between DFTM and the LSS. The interfaces between DFTM and RSIs are uniform, although because of the heterogeneity of the LSSs, the interfaces between RSIs and LSSs are dissimilar.

An RSI also arranges the execution order of subtransactions in the LSS it manages. An RSI enables subtransactions to be executed in the quasi serialization order specified in their parameters. RSIs will be discussed in greater detail in Section 4.

For the sake of simplicity, it is advantageous to conceal from users much of the detail of RSIs. For this purpose, the InterBase System includes an RSI Directory which provides users with location and distribution transparency. This Directory lists those RSIs which may be invoked by users and passes this information to DFTM on request.

### 2.4 Distributed Flexible Transaction Manager

The concept of a flex transaction model [ELLR90] makes possible the correct execution of a transaction in a manner beyond the traditional *all-or-nothing* semantics. The Flex Transaction Model supports flexible transactions, mixed transactions, and time constraints on subtransactions. Unlike the conventional transaction model, the Flex Transaction Model also allows users to decide which of its subtransactions must be committed or aborted in the final stage of the execution of the transaction. In the InterBase System, such transactions are facilitated by the IPL Language and DFTM, which will be discussed in detail in Section 5.

### 2.5 Distributed Concurrency Controller

The InterBase System allows global transactions to execute with a high degree of concurrency. Distributed Concurrency Controller (DCC) determines which global transactions can be executed simultaneously and which must wait until specific conditions are satisfied. This feature is an outgrowth of the principles of transaction grouping, which will be addressed in Section 6. At present, DFTM can execute global transactions for IPL programs on SUN, Sequent, HP, and NeXT workstations. Because DFTM allows several global transactions to be executed simultaneously, the InterBase System can be run in a multi-user environment.

In general, user ease and confidence is promoted by the provision of high level user interfaces,

which may take the form of graphical or object-oriented interfaces. We are currently developing a graphical interface and plan future work on additional interface varieties.

# 3 InterBase Parallel Language

Multidatabases, which have been inadequately served by the traditional transaction models, call for the formulation of more powerful and flexible transaction models. A new *Flex Transaction* Model, developed by the InterBase project, was presented for the first time in [ELLR90, LEB92]. The model supports (1) *flexible transactions*, which can tolerate the failure of individual subtransactions by appreciating that a given subgoal can be reached by the execution of any one of a set of subtransactions; (2) *mixed transactions*, which allow the co-existence of compensatable and non-compensatable subtransactions within a single global transaction; and (3) a *time constraint* on subtransactions, which improves the flexibility of subtransaction scheduling. The relationships among subtransactions take the form of execution dependencies; therefore, the final result of a global transaction may be manifested in more than one set of subtransaction executions. At the final stage of a global transaction, the user must decide which set is preferred. The subtransactions in the desired set are then committed, while other subtransactions are aborted.

The Distributed Operation Language (DOL), an InterBase Language, has been provided to users for application level programming [ROEL90]. With DOL, users are able to define a global transaction and to specify the message passing among its subtransactions. However, DOL cannot provide the environment needed to implement the Flex Transaction Model. For example, DOL cannot specify flexible, mixed, or time-constrained transactions. These limitations led to the design and development of IPL.

Through IPL, the Flex Transaction Model can be mapped into a programming language. IPL is a declarative language, provided with features such as a dependency description, time constraints, guards, and type declarations. The dependency description allows users to specify the explicit dependencies among the subtransactions of a global transaction. Through the use of time constraints, a subtransaction can be executed with a desired begin and/or end time. A guard specifies other conditions for the execution of a subtransaction. Type declarations identify the desired data type of subtransaction output, which can then be incorporated in IPL programs. IPL also provides users with language constructs to define his own commitment protocol, compensatable subtransactions, and non-compensatable subtransactions. Therefore, IPL facilitates the clean and clear specification of flex transactions in a heterogeneous environment.

In this section, we provide an illustrative example of the capability of IPL supporting applications that both conform to and deviate from the Flex Transaction Model.

## 3.1 An Application Example

Consider a professor from Purdue University who wishes to attend a conference to be held at the Sheraton Hotel in San Francisco from Feb. 4 to Feb. 6, 1992. A global transaction for his trip may

consist of the following objectives:

- He should provide information such as his name, the origin and destination of the trip, the departure and return times, and the types and numbers of the credit cards with which he wishes to pay for his airline ticket;
- Airlines companies should be contacted to book a flight ticket;
- The Sheraton Hotel should be contacted for room reservation; and
- Credit card companies should be contacted regarding the payment for a flight ticket.

Let us assume that, for the purpose of this trip, two airline companies (USAir and United) and two credit card companies (VISA and MasterCard) can be involved.

Suppose that the professor has the following preferences:

1. Order a ticket from United only if no ticket is available from USAir, because if he stays at the Sheraton when traveling by USAir, he can triple his frequent-flyer mileage.
2. Reserve a room at the Sheraton only if an airline ticket is available. If no airline ticket is available, he will not attend the conference.
3. Reserve a room at the Sheraton before 3:00 p.m. on Feb. 3, 1992. If a room is not reserved for him before his departure, he will also decide not to make the trip.
4. The cost of the airline ticket should be $\leq$ \$350, this being his maximum budget.

These four objectives can be decomposed as four sets of subtransactions $\{user\}$, $\{usair, united\}$, $\{sheraton\}$, and $\{visa, master\}$, where they are defined as follows:

| | |
|---|---|
| *user*: | Obtain the information from the customer; |
| *usair*: | Order a ticket from USAir; |
| *united*: | Order a ticket from United Airlines; |
| *sheraton*: | Reserve a room at the Sheraton; |
| *visa*: | Pay for the airline ticket with VISA; |
| *master*: | Pay for the airline ticket with MasterCard. |

As indicated by the client's preferences, there are explicit dependencies among subtransactions *usair*, *united*, *sheraton*, *visa*, and *master*, which can be defined as follows: subtransaction *united* will be executed only if the execution of the subtransaction *usair* fails, and subtransaction *sheraton* will be executed only if one of the subtransactions *usair* and *united* succeeds and one of the subtransactions *visa* and *master* succeeds.

If a subtransaction $S_i$ takes the output of another subtransaction $S_j$ as its input argument, we then say that there is an **implicit dependency** between $S_i$ and $S_j$, as opposed to the **explicit dependency** described previously. While subtransactions with explicit dependencies must be executed in a serial mode, one after another, those with implicit dependencies may be executed in pipe mode, in parallel.

There are also implicit dependencies between the set $\{usair, united\}$ and the set $\{visa, master\}$, because if no airline ticket is available, it is unnecessary to pay for it, and if there is insufficient credit

remaining on a credit card, no ticket will be sold. Therefore, the two sets of subtransactions depend upon each other, and implicit dependencies may be used to define this relationship. Because all other subtransactions need the information acquired by subtransaction *user*, there are also implicit dependencies between *user* and the other subtransactions.

It is clear that the global transaction will succeed if one of *usair* and *united* succeeds, *sheraton* succeeds, and one of *visa* and *master* succeeds.

In the next subsection, we will describe the language components of IPL through illustrations drawn from our example.

## 3.2 IPL Components

IPL contains four fundamental components : objects and types, subtransaction definitions, dependency descriptions among subtransactions, and acceptable sets [EC92].

### 3.2.1 Objects and Types

Objects in IPL serve as results of and arguments to subtransactions in an IPL program. Therefore, in IPL, each subtransaction is associated with a type. Types have unique names and are used to categorize objects into sets that are capable of participating in a specific set of subtransactions.

A type specifies the kind of result a successful subtransaction will produce. Because the result of a subtransaction is an object, it can be easily transferred to another context.
For example, to define the type of the output of subtransaction *sheraton* in this section, an IPL format could be:

```
class room of
   customer : charString;
   roomNo: charString;
   cost : real;
   time : charString;
endclass;
```

*room* is used as the name of the type. **class**, **of** and **endclass** are used as keywords in IPL.

### 3.2.2 Definition of Subtransactions

The definition of a subtransaction, with a unique identifier, consists of the following parts: (1) the input parameters, each specified by the name of a subtransaction other than itself; (2) the type of its result; (3) the name of the software system used by the subtransaction; (4) the site on which the subtransaction is executed; (5) time constraints; (6) guards which impose constraints other than those defined in (5); (7) subtransaction operations; (8) a commit operation which is executed when the subtransaction is instructed to commit by its composing flex transaction; (9) an abort operation which is executed when the subtransaction is aborted. Parts (1), (5), (6), (8), and (9) of

a subtransaction are optional. Default values are used if the site and the timeout constraints are not specified.

For example, one can define the context of the subtransaction *sheraton* in this section as follows:

```
subtrans sheraton (user) : room
            use room_reserve at Sheraton_Hotel before Feb 3 15:00 EST 1992
  beginexec
    reserve a room for user.name.
  endexec
  beginundo
    cancel the reserved room for user.name.
  endundo
endsubtrans
```

The output type of subtransaction *sheraton* is *room*, which indicates that an object *room* is returned as its result if the subtransaction succeeds. The local software system involved is the room_reserve System, which runs on the machine named Sheraton_Hotel. The subtransaction is allowed to run before Feb 3 15:00 EST 1992. Otherwise, it will be aborted as if its execution had failed. The result of another subtransaction *user* becomes the input of this subtransaction, providing such user-related information as the name of the customer. The body of the subtransaction operations is defined by the IPL keywords **beginexec** and **endexec** (the construct < *executing* >) and the body of the undo operation by keywords **beginundo** and **endundo** (the construct < *undo* >). The body of the commit operation, which is not defined for this subtransaction, should be defined by keywords **beginconfirm** and **endconfirm** (the construct < *confirm* >).

### 3.2.3 Dependency Description

The dependency description, the third component of IPL, provides users with a mechanism for specifying the explicit dependencies among the subtransactions of a global transaction. That is, the execution order of subtransactions of a global transaction can be defined using the IPL dependency description. For the example, the execution order among subtransactions in the example discussed in this section can be defined:

```
dependency
  not usair : united ;
  (usair or united) and (visa or master) : sheraton ;
  ( 1 : usair, united ) and sheraton and (1: visa, master) : accept ;
enddep
```

The dependency description indicates that subtransaction *united* will be executed only if the execution of the subtransaction *usair* fails, and subtransaction *sheraton* will be executed only if one of the subtransactions *usair* and *united* succeeds and one of the subtransactions *visa* and *master* succeeds. It also indicates that the global transaction will succeed if one of *usair* and *united* succeeds, *sheraton* succeeds, and one of *visa* and *master* succeeds. **accept**, a keyword of

IPL, indicates the succeed/fail status of the global transaction ($GT$). If its value is *true*, then $GT$ succeeds; if *false*, then $GT$ fails; otherwise, $GT$ keeps running until **accept** becomes *true* or *false*.

### 3.2.4 Acceptable Sets

Acceptable sets, the fourth component of IPL, begin with the keyword **acceptable_sets** and end with the keyword **endaccs**. An acceptable set consists of a subtransaction list and a sufficient acceptable condition of the global transaction. When a global transaction reaches its final status, the user is asked to select a preferred acceptable set from an array of choices. All the subtransactions in an acceptable set in the array must be successful. Successful non-compensable subtransactions are maintained in an uncommitted state until the global transaction is completed. When the user chooses an acceptable set and the global transaction commits, the uncommitted subtransactions in the acceptable set then perform their commit operations, all other uncommitted subtransactions perform their abort operations, and the compensatable subtransactions not in the acceptable set perform their compensating operations. When the global transaction decides to abort, all the successful subtransactions perform their abort operations or compensating operations.

For the example discussed in this section, the acceptable sets could be:

**acceptable_sets**
  (*usair*, *sheraton*, *master*), (*usair*, *sheraton*, *visa*),
  (*united*, *sheraton*, *master*), (*united*, *sheraton*, *visa*)
**endaccs**

In this example, four acceptable sets are included; they are subtransaction sets (*usair*, *sheraton*, *master*), (*usair*, *sheraton*, *visa*), (*united*, *sheraton*, *master*), and (*united*, *sheraton*, *visa*). The success of any of these four subtransaction sets will result in the success of the global transaction, thus providing function replication within the global transaction.

## 3.3 An IPL program for the Example

Based upon the IPL components set forth above, an IPL program for the application example discussed in this section could be developed as follows:

```
program
  class creditCard of
    cardholder : charString;
    num : charString;
    creditRemains : real;
    reserved_credit : real;
  endclass;
  class user-info of
    name : charString;
    origin : charString;
    destination : charString;
```

departure_time : **charString**;
return_time : **charString**;
visa : **creditCard**;
master : **creditCard**;
**endclass**;
**class** ticket **of**
flightNo : **charString**;
customer : **user-info**;
cost : **real**;
paid_by : **creditCard**;
**endclass**;
**class** room **of**
customer : **charString**;
roomNo: **charString**;
cost : **real**;
time : **charString**;
**endclass**;
**subtrans** *user* : user-info **use** user_interface **at** Customer_Service
**beginexec**
obtain the information from the customer.
**endexec**
**endsubtrans**
**subtrans** *usair* (*user*, *visa*, *master*) : ticket **use** ticket_order **at** USAir
**beginexec**
reserve a ticket for *user*.name.
**endexec**
**beginconfirm**
**if** *master* is chosed for commit
**then** order the reserved ticket, pay with *master*.
**else** order the reserved ticket, pay with *visa*.
**endconfirm**
**beginundo**
cancel the reserved ticket for *user*.name.
**endundo**
**endsubtrans**
**subtrans** *united* (*user*, *visa*, *master*) : ticket **use** ticket_order **at** United_Air
**beginexec**
reserve a ticket for *user*.name.
**endexec**
**beginconfirm**
**if** *master* is chosed for commit
**then** order the reserved ticket, pay with *master*.
**else** order the reserved ticket, pay with *visa*.
**endconfirm**
**beginundo**
cancel the reserved ticket for *user*.name.
**endundo**

**endsubtrans**
**subtrans** *sheraton* (*user*) : room
use room_reserve **at** Sheraton_Hotel **before** Feb 3 15:00 EST 1992
**beginexec**
reserve a room for *user*.name.
**endexec**
**beginundo**
cancel the reserved room for *user*.name.
**endundo**
**endsubtrans**
**subtrans** *visa* (*user*, *usair*, *united*) : creditCard use cridt_process **at** VISA_Card
*user*.visa.num is valid **and**
((*usair*.cost $\leq$ $350 **and** *user*.visa.creditRemains > *usair*.cost) **or**
(*united*.cost $\leq$ $350 **and** *user*.visa.creditRemains > *united*.cost)) ;
**beginexec**
reserve the credit for the ticket for *user*.name
**endexec**
**beginconfirm**
pay for *usair* if *usair* is chosen for commit.
otherwise, pay for *united* if *united* is chosen for commit.
**endconfirm**
**beginundo**
cancel the reserved credit for the ticket for *user*.name
**endundo**
**endsubtrans**
**subtrans** *master* (*user*, *usair*, *united*) : creditCard use cridt_process **at** MASTER_Card
*user*.master.num is valid **and**
((*usair*.cost $\leq$ $350 **and** *user*.master.creditRemains > *usair*.cost) **or**
(*united*.cost $\leq$ $350 **and** user.master.creditRemains > *united*.cost)) ;
**beginexec**
reserve the credit for the ticket for *user*.name
**endexec**
**beginconfirm**
pay for *usair* if *usair* is chosen for commit.
otherwise, pay for *united* if *united* is chosen for commit.
**endconfirm**
**beginundo**
cancel the reserved credit for the ticket for *user*.name
**endundo**
**endsubtrans**
**dependency**
**not** *usair* : *united* ;
(*usair* **or** *united*) **and** (*visa* **or** *master*) : *sheraton* ;
( 1 : *usair*, *united* ) **and** *sheraton* **and** (1: *visa*, *master*) : **accept** ;
**enddep**
**acceptable_sets**
(*usair*, *sheraton*, *master*), (*usair*, *sheraton*, *visa*),

(*united*, *sheraton*, *master*), (*united*, *sheraton*, *visa*)
**endaccs**
**endprogram**

We will now use the foregoing example to elucidate some of the principles set forth earlier. The example illustrates how the preferences of a user are represented in an IPL program, how the time constraint and guard functions operate, and how explicit and implicit dependency relationships among subtransactions of a global transaction are presented.

Explicit dependencies are relationships among subtransactions of a global transaction which determine their correct execution order. These dependencies must therefore be taken into consideration by the Flex Transaction Model. In IPL, the explicit dependency relationships among subtransactions are defined in the dependency description construct. A subtransaction is eligible to be scheduled only if its dependency becomes *true*. In our example, preferences 1 and 2 can be seen as execution dependencies for *united* and *sheraton*, respectively.

Mixed transactions involving a variety of subtransaction classes can be implemented through the careful definition of an executing partial subtransaction, confirming partial subtransaction, and undoing partial subtransaction for each subtransaction, as illustrated in Section 5.4. In our example, subtransaction *sheraton* is compensatable, while *usair*, *united*, *visa*, and *master* are non-compensatable subtransactions.

The time constraint for each subtransaction is defined in a < *time_expr* > construct. In our example, preference 3 is a time-constraint for subtransaction *sheraton*.

A guard imposes another condition for the execution of a subtransaction. In our example, the guards for *visa* and *master* ensure that the ticket can be paid for by a credit card provided by the customer. Just as the failure of *usair* or *united* will abort the global transaction, so will the failure of *visa* or *master*; guards therefore also guarantee the correct execution of the global transaction.

The usefulness of the concept of implicit dependencies among subtransactions of a global transaction is also illustrated here in this example, there are implicit dependencies between the two groups of subtransactions {*usair*, *united*} and {*visa*, *master*}. There are also implicit dependencies between the subtransaction *user* and all other subtransactions.

Although there are four acceptable sets of subtransactions in this example, subtransactions *usair* and *united* are exclusive [BCC+92], indicating that, at a given time, at most one of them is *true*. There are therefore at most two acceptable sets of subtransactions that can be listed for the user to choose among. Only those acceptable sets which include no *false* subtransactions can be listed for selection. The user must choose one of those acceptable sets as the final result. All the subtransactions in the chosen set can and must be committed, and all other subtransactions must be aborted. The commitment in this example is semantic-based, since different commitment protocols are applied to subtransaction sets {*user*}, {*usair*, *united*, *visa*, *master*}, and {*sheraton*}. The appropriate protocol can be deduced from the semantics of these subtransactions, as illustrated in Section 5.4.

As the foregoing example was intended to illustrate the control structures of the IPL language,

rather than the details of LSSs and RSIs, we have used only pseudo codes for each action. This example particularly highlights the effectiveness of confirming and undoing partial subtransactions, since most businesses allow customers to make, confirm, and cancel reservations without an extra charge within a period of time which is sufficient for the execution of a global transaction.

In summary, the InterBase Parallel Language creates an appropriate environment for the execution of flexible transactions. IPL includes and extends all the functions of the DOL Language [ROEL90], while providing an environment in which global transaction management and query processing are integrated. IPL can be considered a general purpose language, because the text of subtransaction operations is not passed through the IPL interpreter. Moreover, IPL has no knowledge of the LSSs except for an understanding of their operation languages. Therefore, IPL does not violate the autonomy of LSSs. In general, IPL offers both great power and a particular suitability for the execution of global applications in a heterogeneous environment.

Although IPL is designed to support the Flex Transaction Model, it can be used in any heterogeneous environment, as it makes no assumptions about the LSSs on which IPL programs are run. IPL can also be used as a data-flow language, since a subtransaction will not be scheduled before its prerequisite data.

# 4 Remote System Interfaces

The issue most effecting the design and implementation of systems in a heterogeneous environment is that of the heterogeneity of Local Software Systems (LSSs). A uniform system-level interface is necessary in this setting to mask the differences in the details of invoking, input, and output format of individual LSSs. A Remote System Interface (RSI), acting as the interface between an LSS and DFTM, can provide such a uniform interface. RSIs were touched upon in Section 2.3; they will now be examined in detail.

## 4.1 The Definition and the Advantages of RSIs

As defined in Section 2.3, an RSI is a special agent located between DFTM and an LSS. It translates the command text from DFTM into a format understandable by the LSS and sends the translated text to the LSS for execution. It also manages the data flow and data format transformation between DFTM and the LSS. The interfaces between DFTM and RSIs are uniform, although the interfaces between RSIs and LSSs are various.

RSIs therefore provide a uniform system-level interface to the global transactions, while dealing with the heterogeneity of the LSSs. While this enhances the convenience to users, the InterBase System is forced to reconcile the uniformity and heterogeneity of these different levels. RSIs make no assumption about LSSs, so the local autonomy of the LSSs is easily preserved.

An RSI also arranges the execution order of subtransactions in the LSS it manages. By using the grouping triples associated with subtransactions, an RSI enables the subtransactions to be

executed in the quasi serialization order specified in their parameters. Distributed concurrency control is therefore easily implemented through the use of RSIs.

A second advantage of incorporating RSIs into the InterBase System is their simplifying effect on the complexity of DFTM. Because RSIs form a uniform interface between DFTM and LSSs, masking the details of LSSs, DFTM needs no detailed information about the LSS. That is, from the point of view of DFTM, all RSIs appear the same. The RSI enables DFTM and the LSSs to communicate by providing the necessary format transformations.

RSIs are also responsible for preserving the autonomy of each LSS [DEK90]. The autonomy of a system is informally defined as its right to behave according to its design [DELO89]. This requirement precludes the making of any modifications to LSSs. An RSI acts as a proxy user of the LSS it manages, encompassing it in a sort of logical shell. The only underlying LSS interfaces accessible to the RSI are those originally designed to be available to the users of the LSS.

A third advantage of RSIs is the flexibility they permit when a new LSS is added to the InterBase System. In this instance, DFTM remains untouched; we need only to provide an RSI for the LSS and to add this information to the RSI Directory. Similarly, when an LSS is removed from the InterBase System, we need only remove the appropriate information from the RSI Directory; no modification of DFTM is required.

Finally, RSIs effectively support the client/server model for LSSs. That is, by designing an RSI for an LSS, we can then treat the LSS as a server which provides services for subtransactions. This greatly simplifies the complexity of DFTM and adds flexibility to the InterBase System.

### 4.2 RSI Servers and RSI Services

The components of an RSI are a server and its services. An RSI server provides an RSI service to a subtransaction, and the RSI service is used to execute the subtransaction.

Upon receiving an execution request from a global transaction ($G_i$) for one of its subtransactions ($S_j$), an RSI server initiates an RSI service for $S_j$ and makes a connection between $G_i$ and the RSI service. The initiation of the RSI service should be in the quasi serialization order specified in the parameters of the incoming subtransactions. Therefore, RSI servers are also engaged in the implementation of distributed concurrency control over the InterBase System. An RSI server allows several nonconflicting subtransactions for its associated LSS to be executed in parallel so as to achieve a higher degree of concurrency.

After the execution request has been granted, $G_i$ communicates directly with the RSI service to execute $S_j$. The command and input data for $S_j$ provided by $G_i$ is transformed by the RSI service into a format understandable by its associated LSS. The RSI service then sends the transformed command and data to the LSS to execute. When the execution is completed, the RSI service collects its output, performs the necessary format transformation, and sends the output back to $G_i$. The RSI service repeats this procedure until it receives a close command from $G_i$, at which time, it reports to the original RSI server and then exits. The RSI server thus monitors the status of running, completed, and waiting RSI services.

## 4.3 RSI Directory

In the InterBase System, oversight of the locations and capabilities of the various RSIs and their corresponding LSSs is relegated to the RSI Directory. This centralized entity houses a variety of information pertaining to RSIs and LSSs, including the proper channel to use for communication, the supported communication and connection protocols, allowable data transfer methods, and the time zone of the location. Each of these informational categories will contain a wealth of smaller detail. The information regarding communication channels could specify a local area network and an address, a serial line and a device name, or a local program and a path. The communication protocol will usually be a function of the communication channel. Procedures are detailed for making contact with each RSI and its corresponding LSS. These will vary with the characteristics of the RSI; some RSI servers are always running, some are started automatically by their host when requested, and some must be started explicitly before they can be used. Finally, since DFTM allows data to be transferred between subtransactions in either batch mode or pipe mode, the RSI Directory must know which modes are compatible with each RSI and in what manner the data exchange should take place. In summary, the RSI Directory provides a systematic approach for locating and invoking format transparency of LSSs among different networks.

## 4.4 RSI Server Activation

Because RSI servers are not the standard servers of the computer systems on which the InterBase System depends, we have provided a mechanism to allow DFTM to activate an RSI server whenever necessary. RSI servers can either be activated upon demand or left continuously operating.

**Static Activation :** In many cases, it is simplest to assume that the RSI server for a given LSS is always running. In this case, no special measures need to be taken to activate the RSI server when it is needed.

**Dynamic Activation :** Alternatively, DFTM can activates an RSI server whenever needed. The activation method will necessarily be dependent on the operating system platform supporting the RSI server. The advantage of this approach is flexibility.

In the InterBase System, the static activation option is preferred, with the dynamic activation as a default option. The activation information for an RSI is kept in its entry in the RSI Directory.

## 4.5 Interface between RSI Services and Local Software Systems

A uniform system interface is of great importance to the smooth processing of global transactions, since it masks the details of the invoking formats of various LSSs. At the same time, it increases the complexity of RSIs, which must transform the uniform interface into a variety of formats compatible with the standard interfaces of the individual LSSs. The heterogeneity of the different systems is manifested at the point of interface between an RSI service and its LSS.

This interface, which serves to deliver the input to the LSS, collect its output, and monitor its execution, falls within the purview of the RSI service. It will necessarily be very LSS specific,

requiring detailed information about the workings of a particular LSS. The complexity of designing this interface is inversely proportional to the flexibility of the interface design of a particular LSS. Fewer difficulties are presented by an LSS which allows users to select one among multiple input sources and to specify the exact form of its output.

As a simple example, consider a Unix software system such as Ingres, in compare with a PC utility, such as Lotus 123. Ingres lets users specify that a command text and input be obtained from a text file, and the output can be easily redirected to another file. An RSI service can activate an execution of Ingres by combining the command file and input file into a file of its choosing, redirecting it to the Ingres DBMS, and then redirecting the output to another file. When the execution of Ingres completes, all the output is in a file, available for dispatch as needed. Most Unix software systems allow applications to specify their sources of input data and command text and the destination of the output. In this case, the interface between RSI services and LSSs is very easy to design.

On the other hand, software systems written for PCs can present many difficulties for interface design. Such software systems frequently do their own input processing, some to the extent that they have their own keyboard drivers. For such a system, redirecting the command text or input data from a file or serial line can be very difficult, although it is usually possible, given a detailed knowledge of the hardware platform. Capturing the output of these LSSs is similarly difficult. Many insist on writing directly to the screen, using cursor motion commands, complex borders, and flashing text. Although these feature are intended to assist a user, capturing only that part of the output which is useful data can be a challenge.

When designing and implementing an RSI, we should therefore carefully examine the features of the LSS in question with an eye to both efficiency and the preservation of its local autonomy. Clearly, an RSI must be tailored specifically to meet the particular needs of an LSS.

It is also necessary to develop a method of combining the command text and input data, as these must be presented to some software systems as a single unit. The input for a subtransaction usually comes from another subtransaction, while the command text is defined in an IPL program. These components can be consolidated by taking the command text as the main text and inserting the input data within it. The **insert-data** primitive, a pseudo-statement in the command text, triggers the RSI service to replace the primitive with input data taken from the output of the indicated subtransactions. This primitive must also be tailored to the specific software systems, as most have their own input data format and will not accept tabular data.

In RSI design, the foremost consideration must always be the retention the local autonomy of the corresponding LSS. This is preserved by using only the standard interfaces already provided by the LSS.

# 5 Distributed Flexible Transaction Manager

The Distributed Flexible Transaction Manager (DFTM) is at the center of the InterBase System. DFTM interprets and coordinates the executions of all global transactions over the system.

## 5.1 The Operational Principle of DFTM

As the manager of global transactions on the machines that make up the InterBase System, DFTM is a distributed entity. It is also a conceptual entity, because it exists in theory only at those times when there is no global transaction running on the InterBase System. Thus, DFTM provides location transparency for the InterBase System.

For each execution of an IPL program, DFTM generates an image of itself, which is responsible for the consistent and reliable execution of the program. The existence of a DFTM image is therefore coincident with the execution of a global transaction in the InterBase System; after the execution of the global transaction, the DFTM image disappears. By providing a DFTM image for each global transaction, DFTM allows several global transactions to run concurrently. DFTM images must communicate with each other either directly or indirectly to coordinate their executions.

As the interpreter of IPL programs, DFTM is responsible for (1) checking the syntax and semantics of global transactions; (2) managing the flow of control specified by the global transactions; (3) activating and opening connections to RSIs; (4) monitoring the status of the individual RSIs; (5) obtaining DCC permission to execute subtransactions; (6) executing subtransactions (via corresponding RSIs), whenever possible; (7) determining the final status (accepted or unaccepted) of the global transactions; and (8) committing or aborting their subtransactions according to their accepted or unaccepted status.

The DFTM image ($DFTM_i$), therefore, governs the entire life cycle of a global transaction, from inception to completion. After syntax and semantic checks of a global transaction ($T_i$), a simple execution graph, which reflects the dependency relations among all its subtransactions, is built for $T_i$. The $DFTM_i$ for $T_i$ then obtains a group id for $T_i$ and grouping triples for its subtransactions from the Group Manager of DCC. The group id and the grouping triples, to be explained in detail in Section 6, are used by Subtransaction Schedulers of DCC in the individual RSIs to guarantee that subtransactions of $T_i$ are executed in quasi serialization order on each LSS. The $DFTM_i$ then asks the relevant RSIs to approve the executions of subtransactions whose dependency conditions, time constraint, and guard are all satisfied. Upon receiving an approval, the $DFTM_i$ immediately executes the approved subtransaction on the corresponding LSS, via its RSI, until the best reserving state specified in the IPL program is reached or the execution has failed. The $DFTM_i$ then modifies the execution graph. This process continues until $T_i$ reaches its final status. At that point, the $DFTM_i$ commits those subtransactions selected by the user, and aborts those which the user does not want. Throughout, the $DFTM_i$ consults the RSI Directory to determine the interface and data transfer characteristics of the individual RSIs. The completed execution is reported by the $DFTM_i$ to the Group Manager of DCC. DCC thus tracks executing and completed global transactions.

## 5.2 Interface Operation Primitives from DFTM to RSIs

An initial step in designing an integrated system such as that described in this paper is to determine interface operation primitives from DFTM to RSIs. In the InterBase System, we identify the following basic primitives, which form a uniform system interface between those entities.

- **open**

  *parameters:* RSI name, login name, password, grouping triple

  The open primitive determines the existence of the specific RSI server and activates it if it is not already running. It instructs the RSI server to initiate an RSI service for its LSS and to make a connection between the RSI service and the global transaction which issued the open primitive. The RSI service performs any necessary initialization, handles any logging in and user verification, and then waits for further instructions from the global transaction. An open command returns a socket which, upon a successful execution, becomes the communication channel between the global transaction and the RSI service. The RSI service for the subtransaction must be opened before the commands for a subtransaction can be sent to execute. In turn, until the subtransaction is allowed to execute, the execution of the open primitive will be blocked by the RSI server. Therefore, the execution of subtransactions on an LSS will be in the quasi serialization order specified in their group ids.

- **process**

  *parameters:* socket, command text, input data

  The process primitive informs the specific RSI service bound by the socket to accept the command text and input data and to translate them into a form compatible with its LSS. The RSI service then sends translated command text and input data to the LSS to execute. Finally, the RSI service captures and transforms the output of LSS, and sends it back as the return value to the global transaction which issued the process command. The command text can be derived from the executing partial-subtransaction, the confirming partial-subtransaction, or the undoing partial-subtransaction.

- **abort**

  *parameters:* socket

  The abort primitive instructs the RSI service bound by the socket to terminate the processing of the associated subtransaction by its LSS. It will then nullify any effects of the subtransaction. This command is invoked only when an unrecovered error arises during the execution of a global transaction.

- **close**

  *parameters:* socket

  The close primitive is used to terminate the RSI service bound by the socket. Temporary files are purged, the LSS is instructed to exit, the RSI service reports the closing to its RSI servers and to the global transaction, and then the RSI service itself exits.

These four primitives are all considered to be blocked. That is, after issuing one of these four primitives, the execution of the global transaction is blocked until the underlying RSI server or RSI service returns a value or an error message. Because we assume that communication between global transactions and all RSIs is uniform in format, arguments to the commands must be designed

to accommodate a variety of LSS types. By using sockets as the communication channel between global transactions and RSIs, UNIX file operation primitives are used to support the communication between global transactions and RSIs. This greatly reduces the complexity of DFTM and simplifies the detection of RSI failures. When an operation to a socket returns error, DFTM concludes that the corresponding RSI has crashed.

### 5.3 Data Transfer Methods between Subtransactions

In IPL, the output of a subtransaction can become an input parameter for another subtransaction. An important function of DFTM is the proper transfer of data between subtransactions. Two data transfer methods present themselves as options.

In the first approach, two subtransactions which have an implicit dependency can be executed sequentially. Upon the completion of the first subtransaction, its output is transported to the second subtransaction, which is then run with that data as input. The total execution time for this process is roughly the sum of the execution times of each subtransaction, the data transfer time, and the transfer protocol startup time. Because the abortion of one subtransaction will not influence the other, consistency is easily maintained. This approach is therefore preferred in multidatabase environments.

The second method executes the two subtransactions in parallel. Here, fragments of data must be transported from the first subtransaction to the second as soon as they become available. In many cases, this can be accomplished by the stream transport protocols supported by the host computer. The parallel execution of subtransactions has a higher degree of concurrency than does sequential execution, while also permitting the execution of subtransactions to be controlled dynamically. The commitment or abortion of subtransactions can also be controlled, as will be discussed in the following Section. On the other hand, with parallel execution, the failure of one subtransaction may mandate the abortion of the other. Cascading abortions may occur; therefore, this method calls for careful handling.

DFTM is capable of selecting one of these transport method options for each subtransaction specified in a global transaction. As this decision is made wholly by DFTM, both the input and output transport are specified in the commands sent to the RSIs.

### 5.4 The Semantic-Based Commitment Protocol

The InterBase System implements commitment through a Semantic-based Commitment Protocol. Commitment is conventionally regulated by the system, but DFTM places the control of commitment in the hands of users. With the three IPL language constructs $< executing >$, $< confirm >$, and $< undo >$, users can define their own commitment protocols based on the semantics of the subtransactions.

Let us now introduce a concept fundamental to the commitment protocol, that of **reserving** states. From a reserving state, a subtransaction can easily be undone without the creation of any inconsistencies. For example, in an airline reservation system, one can either reserve or order a

ticket. The reserving process requires that one must later confirm one's reservation (i.e., order the ticket within a specific time period after the reservation), but there is no penalty for cancellation. In contrast, cancellation of an ordered ticket is either impossible or carries a penalty. "*A ticket is reserved*" is therefore an example of a reserving state for an ordering air-ticket subtransaction.

From its definition, we can conclude that a reserving state can be the state of a subtransaction just before its execution, the state of a reversible subtransaction just after its execution, and any state of a read-only subtransaction. Therefore, a subtransaction may have several reserving states. However, only one among them will not lead, upon the execution of the subtransaction, to any other reserving state. We define this as the **best reserving** state of the subtransaction. For example, for an ordering air-ticket subtransaction, "a ticket is reserved" is the *best reserving* state for that subtransaction. It is clear that any subtransaction has only one best reserving state.

The best reserving state forms an axis which is critical to the definition of the **executing partial-subtransaction** (EPS), the **confirming partial-subtransaction** (CPS), and the **undoing partial-subtransaction** (UPS). For a given subtransaction, the EPS includes all operations from the beginning to the best state, the CPS those from the best state to its commitment, and the UPS those from the best state to its abortion. For example, for an ordering air-ticket subtransaction, "reserve a ticket" is the EPS, "order the reserved ticket" the CPS, and "cancel the reserved ticket" the UPS.

In IPL, an $< executing >$ construct indicates an EPS, a $< confirm >$ construct a CPS, and an $< undo >$ construct a UPS.

The best reserving state arises from the semantics, rather than from the syntax, of a subtransaction. It is therefore the user's responsibility to define the best reserving state for a subtransaction, and, by extension, the pertinent EPS, CPS, and UPS. This has promoted us to term this method the *Semantics-based Commitment Protocol.*

Let us apply this discussion to the determination of the EPS, CPS, and UPS for several types of subtransactions. For a compensatable subtransaction ($S_i$) [GM83, KLS90], the best reserving state is, by definition, the state after the execution of $S_i$. Therefore, for $S_i$, the EPS is $S_i$ itself, and the UPS could be the compensating subtransaction for $S_i$. No CPS needs to be defined for $S_i$.

If $S_i$ is a read-only subtransaction, or if the data modified by $S_i$ need not be consistent, then only the EPS for $S_i$ must be defined. Clearly, in this case, the EPS is $S_i$ itself.

The situation is more complex for a non-compensatable subtransaction ($S_j$) of a global transaction ($T_i$). After the EPS of $S_j$ reaches the best reserving state, two choices are presented to the user in defining the next step of the EPS and the ensuring executions of the CPS and UPS. The result of this choice generates different definitions for EPS, CPS, and UPS in the two instances.

- If $S_j$ must be run in isolation, the EPS commits at the best reserving state of $S_j$. During the final stage of the execution of $T_i$, the user chooses whether to commit or abort $S_j$. In the former case, the CPS is issued to bring $S_j$ to completion and to commit it; in the latter, the UPS is issued to undo the execution of the EPS.
- If $T_i$ and $S_j$ are capable of intercommunication and need not be isolated completely, at the best reserving state of $S_j$, after reporting the success information to $T_i$, the EPS waits for a

commit/abort signal from the communication channel set between the EPS and $T_i$. Again, during the final stage of the execution of $T_i$, the user chooses whether to commit or abort $S_j$. In the former case, the CPS is issued to send the "commit" signal to the EPS, thus triggering the EPS to commit; in the latter, the UPS is issued to send the "abort" signal, triggering the EPS to abort. This is very similar to the Two Phase Commitment Protocol.

Regardless of the number of partial subtransactions defined for a subtransaction ($S_i$), IPL always execute its EPS first. At the final stage of the global transaction, if $S_i$ must be committed and its CPS is defined, the CPS is submitted for execution to commit $S_i$; on the other hand, if $S_i$ must be aborted and its UPS is defined, the UPS is submitted for execution to undo $S_i$. Therefore, although the semantic structure of IPL is delegated to users for decisions regarding commitment (for each subtransaction, the user defines the best reserving state, as well as the EPS, CPS, and UPS, when necessary), IPL can determine from the definition of a subtransaction the most appropriate type of commitment operation.

In this approach, the commitment protocol for a global transaction is user-defined, based on the semantics of the subtransaction, and is grounded on the commitment protocols of the LSSs. Such a commitment protocol is both flexible and simple, and can be applied to a variety of application environments. The application example in Section 3 illustrates the effectiveness of the Semantics-based Commitment Protocol.

In addition to the commitment protocol just outlined, the implicit dependencies between subtransactions may also be used to control the execution of subtransactions. If two subtransactions $S_i$ and $S_j$ are executed in a pipe mode, and the output of $S_i$ becomes an input parameter of $S_j$, then by generating different data as its output, $S_i$ can control the execution of $S_j$. Furthermore, $S_i$ can use its output to reveal its execution status to $S_j$. For a subtransaction ($S_i$) of a global transaction ($T$), two specially designed subtransactions ($S_s$, $S_t$) can be put in place to control the execution and indicate the status of $S_i$. The output of $S_s$ is used as an input parameter of $S_i$, while the output of $S_i$ becomes the input parameter of $S_t$. Since $S_s$ and $S_t$ are specially designed, $T$ can control their executions and directly access their data. When these three subtransactions are executed in a pipe mode, $S_s$ allows $T$ to control the execution of $S_i$, while $S_t$ reveals the status of $S_i$ to $T$. More specifically, when $S_i$ reaches its prepare-to-commit status, it sends the message *ready-to-commit* as its output to $S_t$ and then waits for input data from $S_s$. $S_t$ is specially designed to allow $T$ to access this message. After collecting such messages from all its subtransactions or running out of time, $T$ then asks $S_s$ to send a *commit* or *abort* message to $S_i$, triggering $S_i$ to commit or abort. A variety of commitment protocols can be implemented in this manner.

# 6 Distributed Concurrency Control

Problems of concurrency control in heterogeneous environments are exacerbated by the autonomy of individual software systems [BK91]. Many software systems incorporate their own concurrency controllers to ensure that local transactions are run in a serializable fashion. Improper synchronization can nevertheless take place at the global level; examples appear in [DELO89]. Each of the

software systems in a heterogeneous environment sees itself as isolated and has no knowledge of the existence of other software systems. The resulting synchronization problems must be addressed by controlled execution and commitment of subtransactions.

Serializability is the traditional correctness criterion for maintaining the consistency of a system where multiple interrelated operations are executed in parallel. In an autonomous environment, however, it is unreasonable to expect to guarantee such a condition. Serializability is predicated on a degree of knowledge of the execution of local operations which is incompatible with the preservation of local autonomy [DEK90, DELO89, GMK88]. The theory of Quasi Serializability which the InterBase project has developed provides an effective alternative [DE89, ED91]. It defines a weaker consistency criterion than the traditional serializability, verifiable in a heterogeneous environment, yet strong enough to guarantee a degree of correctness.

In the current paper, we introduce a Distributed Concurrency Controller (DCC) which is responsible for controlling concurrency among global transactions in the InterBase System. Before executing, a global transaction obtains from the DCC a group id and grouping triples for each of its subtransactions (see Section 6.1 for details). When it wishes to execute a subtransaction, the global transaction issues an open primitive to the subtransaction scheduler in the corresponding RSI. It then becomes the responsibility of the individual RSIs to guarantee that subtransactions on their associated LSSs are executed in the quasi serialization order [DE89, ED91] specified in their group ids. As the DCC requires no information about local systems, local autonomy is preserved; while its decentralized nature permits higher concurrency.

The proposed DCC maintains serializability if local executions are rigorous [BGRS91] and serializable and maintains quasi serializability in all instances.

## 6.1 The Operational Principles of the DCC and Grouping Triples

The high degree of concurrency achieved by the proposed DCC necessitates much more complicated data structures and algorithms than those set forth in [DEK91]. Nevertheless, as the following example illustrates, the advantages of the DCC outweigh the drawbacks of this increased complexity.

Consider a distributed system (DS) consisting of five LSSs: $S_1$, $S_2$, $S_3$, $S_4$, and $S_5$. Let $T_1$, $T_2$, and $T_3$ be global transactions in the DS:

$T_1$ accesses $S_1, S_2$, and $S_3$ in the specified order; i.e, $T_1 = \{T_{1,1}, T_{1,2}, T_{1,3}\}$;

$T_2$ accesses $S_2$ and $S_4$ in the specified order; i.e., $T_2 = \{T_{2,2}, T_{2,4}\}$;

$T_3$ accesses $S_5$, $S_4$, and $S_3$ in the specified order; i.e., $T_3 = \{T_{3,5}, t_{3,4}, T_{3,3}\}$;

where $T_{i,j}$ indicates the subtransaction of global transaction $T_i$ on LSS $S_j$.

The access graphs [DEK91] of $T_1, T_2$, and $T_3$ and the access graph of the first two global transactions (AU) are shown in Figure 2.

Because the graph AU is acyclic, $T_1, T_2$ can be in the same group, such as group 1. However, if $T_3$ is added to AU, a cycle will be created through the connections between $S_3$ and $S_4$ and between $S_4$ and $T_5$. $T_3$ must therefore be in another group (group 2).

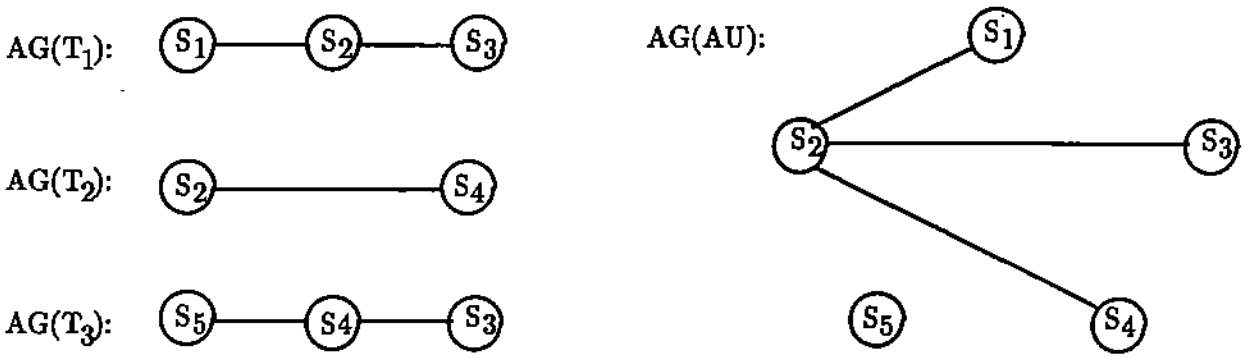


Figure 2: The Access graphs

By the GCC proposed in [DEK91], $T_3$ must be executed after the execution of $T_1$ and $T_2$. If we assume the execution of each subtransaction to take one second, ignoring overhead, then the three transactions take six seconds to complete, with the execution order:

$(T_{1,1}\ T_{2,2})$, $(T_{1,2}\ T_{2,4})$, $T_{1,3}$, $T_{3,5}$, $T_{3,4}$, $T_{3,3}$.

$(T_{i,j}\ T_{k,l})$ indicates that $T_{i,j}$ and $T_{k,l}$ can run at the same time, while $T_{i,j}$, $T_{k,l}$ indicates that $T_{k,l}$ must be executed after the execution of $T_{i,j}$.

The local execution of the subtransactions is follows:

$S_1$: $T_{1,1}$.

$S_2$: $(T_{2,2}\ T_{1,2})$

$S_3$: $T_{1,3}$, $T_{3,3}$.

$S_4$: $T_{2,4}$, $T_{3,4}$.

$S_5$: $T_{3,5}$.

Although, from the global point of view, $T_3$ conflicts with $T_1$ and $T_2$, from a local viewpoint, $T_3$ conflicts only with $T_1$ at $S_3$ and with $T_2$ at $S_4$. Therefore, the execution order

$(T_{1,1}\ T_{2,2}\ T_{3,5})$, $(T_{1,2}\ T_{2,4})$, $(T_{1,3}\ T_{3,4})$, $T_{3,3}$

is also acceptable and requires only four seconds to complete. This example illustrates that local rather than global solutions to conflicts can achieve much greater concurrency. Each subtransaction must be submitted with enough information to allow the Subtransaction Schedulers (STSs) to execute the subtransactions in quasi serialization order. More specifically, the STS for an LSS must know how many subtransactions from each group are on the LSS, as well as the order of groups. This information is obtained in the following manner:

Each request to execute a subtransaction is sent to its STS with a parameter *grouping triple*:

its group id ($GID$);

the id of the transaction group it follows ($TGIF$); and

the number of subtransactions on the LSS in group $TGIF$ ($NTGIF$).

Therefore, when a new open primitive arrives for a subtransaction, the STS knows that the group $TGIF$ is followed by the group $GID$, as well as the number of subtransactions on the LSS in

the group $TGIF$. With this information, the STS is able to guarantee that the executions of subtransactions are in quasi serialization order, while still allowing the highest degree of concurrency.

For the example discussed above, the grouping triples of the subtransactions are illustrated in Figure 3.

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ |
|---|---|---|---|---|---|
| $T_1$ | (1, 0, 0) | (1, 0, 0) | (1, 0, 0) | | |
| $T_2$ | | (1, 0, 0) | | (1, 0, 0) | |
| $T_3$ | | | (2, 1, 1) | (2, 1, 1) | (2, 0, 0) |

Figure 3: The grouping triples of subtransactions

The group 0 is a pseudo group for testing the initial condition. A subtransaction which follows group 0 is in the first group on an LSS and can be executed immediately.

For example, the subtransaction $T_{3,4}$ has the grouping triple (2, 1, 1). From the grouping triple, we know the subtransaction is in group 2; it follows group 1, which has one subtransaction on $S_4$. We therefore know that, after the subtransaction in group 1 completes its execution on $T_4$, there are no more subtransactions from group 1 on $S_4$. We also know that the subtransactions in group 2 are those that can be executed on $S_4$. We also find that, since $T_{3,5}$ has the grouping triple (2, 0, 0), it can be executed immediately, regardless of the status of $T_1$ and $T_2$. The simultaneous execution of global transactions in different groups enables the achievement of a greater degree of concurrency.

With grouping triples as a basis, the distributed algorithms for the DCC can be easily developed.

## 6.2 Group Manager

The Group Manager is the central entity of the Distributed Concurrency Controller. Before execution, a global transaction sends its access graph to the Group Manager. Then, upon receipt of a starting request, the Group Manager provides the global transaction with the id of its inclusive group, along with grouping triples for all its subtransactions. The Group Manager also processes the termination requests of global transactions.

Global transactions that form an acyclic access graph are grouped together. The Group Manager adds global transactions to a group until a cycle is formed. A new group is then created, and incoming transactions are added until a new cycle is formed; the process repeats.

The Group Manager gives a global transaction its group id, as well as grouping triples for all its subtransactions on the basis of the data structures $AGG$, $CGID$, and $SL$ and the procedures *GrantGroup* and *TerminateTrans*. These data structures and procedures are defined as follows:

*AGGs*: the access graphs, each representing a group of global transactions.

$CGID$: the index to the access graph of the current group. For simplicity, group ids are natural numbers in ascending order. The current group is, therefore, the group with the highest group id.

*SL*: the Site List, each entry of which represents an LSS. Entries hold such information as the number of subtransactions on the LSS in the latest group (*NSLG*) and in the previous group (*NSPG*), as well as the id of the latest group (*LGID*) and of the previous group (*PGID*). The latest group on an LSS is defined as the transaction group with the highest id; it may be the same as *CGID*. Similarly, the previous group on an LSS is defined as the transaction group with the second highest id.

These data structures lead naturally to the implementation of *grouping triples*. The Group Manager, in processing the starting request of a global transaction, provides its subtransactions with grouping triples containing *TGIF*s and *NTGIF*s which are the same as the *PGID*s and *NSPG*s for the associated LSSs.

For the example introduced earlier, after the Group Manager has processed the requests of $T_1$ and $T_2$, the Site List of the system would appear as illustrated in Figure 4.

| | LGID | NSLG | PGID | NSPG |
|---|---|---|---|---|
| $S_1$ | 1 | 1 | 0 | 0 |
| $S_2$ | 1 | 2 | 0 | 0 |
| $S_3$ | 1 | 1 | 0 | 0 |
| $S_4$ | 1 | 1 | 0 | 0 |
| $S_5$ | 0 | 0 | 0 | 0 |

Figure 4: The Site List after Group 1

After the Group Manager has processed $T_3$, the Site List of the system appears as shown in Figure 5.

| | LGID | NSLG | PGID | NSPG |
|---|---|---|---|---|
| $S_1$ | 1 | 1 | 0 | 0 |
| $S_2$ | 1 | 2 | 0 | 0 |
| $S_3$ | 2 | 1 | 1 | 1 |
| $S_4$ | 2 | 1 | 1 | 1 |
| $S_5$ | 2 | 1 | 0 | 0 |

Figure 5: The Site List after Group 2

A comparison of Figure 3 with Figure 5 shows that they are closely related. The grouping triple of the subtransaction $T_{3,4}$ is taken from items 1, 3, and 4 of row 4, Figure 5.

The procedure *GrantGroup*, illustrated in Figure 6, takes the access graph of a global transaction (*AG*) as its argument. The output it generates includes a group id for the global transaction and a list of pairs of (*TGIF*, *NTGIF*) for those LSSs that the global transaction accesses (*LP*). From

this output, grouping triples for its subtransactions are easily developed.

```
procedure GrantGroup(AG)
    if NoCycle(AG, CGID)
    then PutItToGroup(AG, CGID) ;
    else CGID ← CreateANewGroup(CGID) ;
         PutItToGroup(AG, CGID) ;
    endif
    get LP from SL ;
    return CGID and LP ;
endprocedure GrantGroup
```

Figure 6: Procedure *GrantGroup*

In the above scenario, the procedure *NoCycle* takes as its argument the access graph of a global transaction ($AG$) and current group id ($CGID$). It determines whether the addition of the $AG$ to the previous acyclic graph represented by $CGID$ results in a cycle. If so, *true* is returned; otherwise, *false* is returned.

The procedure *PutItToGroup* adds the $AG$ to the group bearing the group id $CGID$. It also adds 1 to the $NSLG$ for those entries of $SL$ representing the LSSs accessed by the global transaction.

The procedure *CreateANewGroup* creates a new group structure. As a new group is created, it copies the $NSLG$ to $NSPG$ and the $LGID$ to $PGID$, setting the $NSLG$ to 0 and the $LGID$ to $CGID$ for those entries of $SL$ representing the LSSs accessed by the global transaction.

The procedure *TerminateTrans*, illustrated in Figure 7, takes the access graph of a global transaction ($AG$) and its group id ($GID$) as the argument and purges the $AG$ from the group of the global transactions represented by $GID$. If the group $GID$ is empty, the group is removed. All the global transactions must call this procedure before they terminate.

```
procedure TerminateTrans(AG, GID)
    PurgeFromGroup(AG, GID) ;
    if IsEmpty(GID)
    then Remove(GID) ;
    endif
endprocedure TerminateTrans
```

Figure 7: Procedure *TerminateTrans*

Because no two access graphs in a transaction group can be the same (this would create at least one cycle), the $AG$ and $GID$ can be used as an id for a global transaction.

Initially, $CGID$ is set to 0, as are the $NSLG, LGID, NSPG$, and $NSPG$ for each LSS. Group 0 is used as the pseudo group for testing the initial condition; that is, a subtransaction which follows

group 0 is in the first group sent to an LSS and can be executed immediately.

### 6.3 Subtransaction Scheduler

As an important component of the RSI server for an LSS, a **Subtransaction Scheduler** (STS) guarantees that the execution of subtransactions on the associated LSS are in the quasi serialization order specified in their group ids. For this purpose, a subtransaction can be viewed as a consistent and reliable execution of an RSI service and its corresponding LSS. Before executing a subtransaction, a global transaction seeks permission by issuing an open primitive to the corresponding RSI. The individual STS then decides when to allow the global transaction to execute this subtransaction. For simplicity, the $GID$ of a global transaction is used as the $GID$ of all its subtransactions.

The STS for each LSS encompasses the data structures *ExecGroup*, *DGroups*, and *GroupInfo* and the procedures *SubtarnsGrant* and *GrantNextGroup*. These features execute and monitor subtransactions on the LSS, as follows:

*ExecGroup*: the $GID$ of the subtransactions group which is permitted to execute on the LSS.

*DGroups*: the set of all subtransactions with a $GID$ different from *ExecGroup*.

*GroupInfo*: information regarding the total number ($NSubs$) and number of terminated ($TSubs$) subtransactions for each group.

```
procedure SubtarnsGrant()
    while true do
      Listen to the SUBIN and SUBDONE ports simultaneously ;
      if a request comes from SUBIN port
      then ExecReq ← the request from SUBIN ;
           (NSubs of GroupInfo[TGIF of ExecReq]) ← (NTGIF of ExecReq) ;
           if GID of ExecReq = EexcGroup or ExecGroup = 0
           then grant the execution of ExecReq ;
                ExecGroup ← GID of ExecReq ;
           else DGroups ← DGroups ∪ ExecReq ;
           endif
      else OutSubs ← the request from SUBDONE ;
           (TSubs of GroupInfo[GID of OutSubs])++ ;
           if (TSubs of GroupInfo[GID of OutSubs]) = (NSubs of GroupInfo[GID of OutSubs])
           then GrantNextGroup(GID of OutSubs) ;
           endif
      endif
    enddo
endprocedure SubtarnsGrant
```

Figure 8: Procedure *SubtarnsGrant*

The procedure *SubtarnsGrant*, illustrated in Figure 8, monitors the execution of subtransactions

in the group *ExecGroup* and processes the incoming subtransaction execution requests. It either allows a subtransaction to be executed immediately (if its *GID* equals *ExecGroup* or *ExecGroup* equals 0) or delays its execution (if its *GID* does not equal *ExecGroup*).

*SUBIN* is the input port which receives incoming subtransaction execution requests from global transactions. *SUBDONE* is the port which receives the termination requests of the subtransactions which have been permitted to execute by the RSI.

If the *NSubs* of a group ($G_i$) equals the *TSubs* of $G_i$, the subtransactions in $G_i$ on the LSS have all been completed. Therefore, we can immediately execute those subtransactions which are in the group that follows $G_i$.

The procedure *GrantNextGroup*, illustrated in Figure 9, removes and then grants all the execution requests in the *DGroups* whose *TGIF* equals the argument of the procedure.

```
procedure GrantNextGroup(GID)
    for each ExecReq in DGroups do
      if TGIF of ExecReq = GID
      then remove ExecReq from DGroups ;
           grant the execution of ExecReq ;
           ExecGroup ← GID of ExecReq ;
      endif
    enddo
endprocedure GrantNextGroup
```

Figure 9: Procedure *GrantNextGroup*

Initially, *ExecGroup* is set to 0, *DGroups* is set to empty, and *NSubs* and *TSubs* for each entry in the *GroupInfo* structure are set to -1 and 0, respectively.

After this detailed examination of the data structures and algorithms of the Distributed Concurrency Controller, let us summarize its main features and advantages:

- **Preservation of Local Autonomy**

  Local autonomy is preserved because the DCC requires no information from and makes no assumptions about LSSs (other than the serializability of local execution).

- **Freedom from Global Deadlock**

  The proposed algorithms are pessimistic, in that they control submissions and executions of the subtransactions of global transactions in such a manner as to avoid undesirable situations.

- **Freedom from Global Livelock**

  A global transaction can be added to the current group (CG) only if its addition will not form a cycle in the CG. If the addition will form a cycle, a new CG is created and no further

global transactions will be added to the original CG. Therefore, global transactions will never be delayed indefinitely.

- **Higher Degree of Concurrency**

  Schedulers based on quasi serializability provide a higher degree of concurrency than those based on serializability, because they allow both serializable and non-serializable executions and can produce a more inclusive set of schedules. While the schedules based on quasi serializability apply even if no information is available about local executions, those based on serializability do not.

  The DCC introduced in this paper provides an even higher degree of concurrency than that presented in [DEK91], because decisions regarding the execution of subtransactions are made by RSIs, not DFTM. Therefore, two global transactions in different transaction groups may be executed in parallel, as long as their subtransactions are executed in the quasi serialization order defined by the group ids on the LSSs. This feature was illustrated by the example in Section 6.1.

- **Prevention of Unnecessary Transaction Abortion**

  The DCC does not abort global transactions if there are inconsistencies among their local executions, because these global transactions are always executed sequentially. Transactions that are executed concurrently have no conflicting local executions and therefore cannot create inconsistencies.

# 7 An InterBase System at BNR - A Case Study

Like many companies with distributed operations, the computing environment at BNR Inc. is a conglomeration of heterogeneous software packages and hardware platforms. The BNR network includes mainframes and workstations scattered across the U. S., Canada, Great Britain, and Japan. As computing resources have been distributed to the various BNR sites, a heterogeneous network of self-controlled or autonomous computing services has arisen. Those services, designed to meet specific database, reporting, analysis, and computational problems, were not thought to be beneficial in tackling problems outside their original purview. Consequently, although many computing services exist, there usually is a lack of integration and accessibility. Knowing what applications are available, what computers house the applications, and how to access these applications is a formidable task.

In response to problems typically associated with such environments, the InterBase System has been installed at BNR Inc. The InterBase System allows users to develop and execute programs which access several applications and computer platforms transparently.

BNR anticipates realizing two major benefits from utilizing the InterBase System. The first benefit is cost reduction. Since the InterBase System can reduce the need for global databases by obtaining data from its local sources, fewer resources are dedicated to disk space and database

administration. The second benefit is convenience; users can now access and process data from otherwise inconvenient or disjointed data sources.

Logically, the InterBase System at BNR conforms to the model illustrated in Figure 1. Physically, DFTM and the Group Manager of DCC reside on SUN and HP workstations. In addition, there are eight RSIs installed on five computer platforms, which consist of UNIX workstations and mainframes running CMS.

## 7.1 Challenges

Several challenges were confronted while installing the InterBase System at BNR. For example, the InterBase System had not previously been installed on a mainframe under CMS. As mainframes employ an atypical C compiler (Whitesmiths LTD's C Language for System/370), correct compilation of the codes proved to be a complex process. A formidable challenge was presented by the fact that only one application can run within a virtual machine (VM) at a given time. While the RSI must run continuously in order to detect an incoming request for service, if the VM is constantly engaged in looking for requests, it cannot at the same time execute the service it is being asked to perform. This problem was approached by having the RSI build an "exec" from the incoming request and executing the service from within the RSI. Yet another hurdle lay in the distinctions between C compilers among different UNIX environments. The InterBase System had originally been developed on a different brand of workstation than that used at BNR. Finally, it was found that BNR's mainframes use a communication protocol, Knet, which is different from what was used in the InterBase Lab at Purdue University. It was necessary to thoroughly analyze the Knet protocol and to then write the appropriate assembly language programs. After establishing communications through these programs, the communication calls were incorporated in the C programs.

## 7.2 Functionality

Despite the small number of RSIs installed on the InterBase System at BNR, the primary functions of database access and data analysis are effectively provided. The InterBase System at BNR lets users develop global applications, including both database applications and data analysis applications, on workstations that access mainframes and other workstations.

The system allows users to operate through a workstation to perform mainframe database queries and capture the resulting output. The query may access data on any other node. For example, a user on node A may have a file of values, such as last names, on node B, which map to a database key field on node C. The user may enter a query, such as "*select projects where manager* =", on node A, to which the results should also be sent. The system thus learns the location of the database and the file of values, merges the query with the file of names to generate the complete query, and sends it off to the database on node C. The results of the query are then sent back to node A.

Data analysis is the other important capability of the InterBase System at BNR. Data are accumulated in databases on many nodes and in data sets at different locations. Data capture is

an integral part of data analysis. With the InterBase System, data capture and analysis can be accomplished in one global application, as illustrated in the following IPL program :

```
program
  subtrans S1 : string use shell at nodeB
    beginexec
      cat /users/$HOME/search.values
    endexec
  endsubtrans
  subtrans S2 (S1) : string use DBS at nodeA
    beginexec
      FIND ACTIVITIES WHERE ACTIVITY = !
      SET FIELD ORDER CURRENT ACTID SYSTEM PROJECT STATUS
      LIST ALL
      BYE
    endexec
  endsubtrans
  subtrans S3 (S2) : string use SAS at nodeC
    beginexec
      options nocenter;
      input actid $ 1-6 system $ 8-14 project $ 16-25 status $ 27-30;
      proc sort; by project status;
      proc means; freq status;
      proc print noobs;
      run;
    endexec
  endsubtrans
  dependency
    S1 : S2 ;
    S2 : S3 ;
  enddep
  acceptable_sets
    (S1, S2, S3)
  endaccs
endprogram
```

The program executes several tasks. First, a file of values to search against is obtained from node B. Second, a query is generated on node A, comprised of DBS query language statements incorporated with the file of values produced at node B. This query is sent to the DBS database on the same node. Third, an analysis program in SAS is activated on node C, using the results returned from the second step; thus in turn generates a report of the results. All these steps are performed automatically within one IPL program.

The success of the InterBase System has encouraged BNR to pursue a broader development, encompassing more varieties and greater sophistication of applications, an increased number of plat-

forms to which the system has been ported, and availability to a greater array of departments. For example, Quality Department has initiated development of a database which collects information from several databases at different sites. The InterBase System will assist in initially populating the database and then periodically updating it, checking the data for accuracy and consistency. An RSI will allow access to this database by other users. When completed, this project will simplify the retrieval of data from diverse databases on separate nodes and will provide a clean, consistent interface to users.

RSIs for other databases within BNR are being considered. These RSIs will allow users to simultaneously query several databases located on separate nodes. They will also have the capacity to simultaneously update multiple autonomous databases, the functions which are more complex than read only functions. Several other projects will also be addressed through the InterBase System; results will be presented in future reports.

# 8 Future Research Directions and Conclusions

The InterBase System is an ongoing multidatabase project. It integrates transaction management and query processing to meet the needs of various global applications over a distributed, autonomous, and heterogeneous environment. IPL, its expressive vehicle, is a low level language and an interactive user-friendly graphical interface is therefore part of our future plan. This graphical approach will render the syntax of individual local applications highly transparent to the user. An object-oriented user interface, being developed, will provide an object-oriented SQL interface to the InterBase System. The recovery mechanism for the InterBase System is also implemented. We also plan to extend the InterBase System to incorporate more machines and operating system platforms, thus serving a larger community of users. The performance evaluation of the InterBase System is also being undertaken.

This paper has addressed the problems inherent in an environment consisting of distributed, autonomous, and heterogeneous software systems. Such an environment is often the natural result of the shifting priorities and needs of an organization as it acquires new hardware and software. Each new system, although it may solve or facilitate a short-term requirement, increases the complexity of global applications which access data and services from several different systems. The solution proposed by the InterBase project is the design of a uniform system interface which will allow a user to write global applications over such an environment. In this paper, the architecture of the InterBase System, Remote Systems Interfaces, and the Distributed Flexible Transaction Manager as well as the interrelationships among these subsystems are described. We also propose the specification of a global application language (IPL), an execution environment for that language, a distributed concurrency controller to guarantee correct interaction between concurrent global accesses, a commitment protocol to ensure global transactional semantics, and a prototype implementation built over a small set of heterogeneous systems. The uniqueness of the InterBase System lies in the IPL language, the semantics-based commitment protocol, the distributed concurrency control, the distributed flexible transaction management, and the preservation of local autonomy.

The proposed InterBase Parallel Language requires no knowledge of the local software systems except their operation languages; therefore, IPL does not violate local autonomy. Furthermore, the InterBase System does not request any information from and makes no assumptions about the local software systems and thus also preserves the local autonomy requirements. Similarly, the InterBase System allows users to invoke several systems and to read/write several databases in a systematic way without violating local autonomy of those software systems. By contrast, most other systems of this type support read-only applications.

Although the InterBase System is far from perfected, we strongly believe that it has the potential to offer an innovative and effective solution to the problems of heterogeneity integration, preservation of local autonomy, and proper execution of global applications. This solution has been shown at BNR Inc. Future research will further demonstrate the benefits inherent in the Interbase System.

## References


[BCC+92] O. Bukhres, Jiansan Chen, Jindong Chen, A. Elmargarmid, Yungho Leu, and Gang Zhu. IPL : The InterBase Parallel Language. In *Proc. of the 2nd International Workshop on Research Issues on Data Engineering : Transaction and Query Processing*, 1992.

[BGRS91] Y. Breitbart, D. Georgakopolous, M. Rusinkiewicz, and A. Silberschatz. On Rigorous Transaction Scheduling. *IEEE Transactions on Software Engineering*, 9, 1991.

[BK91] N. S. Barghourti and G. E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 1991.

[BOT86] Y. Breitbart, P.L. Olson, and G.R. Thompson. Database Integration in a Distributed Heterogeneous Database Systems. In *Proceedings of the International Conference on Data Engineering*, pages 301–310, Los Angeles, CA, February 1986.

[Chu90] C.W. Chung. DATAPLEX: An Access to Heterogeneous Distributed Databses. *Communications of ACM*, 33(1):70–80, January 1990.

[DE89] W. Du and A. Elmagarmid. Quasi Serializability: a Correctness Criterion for Global Concurrency Control in InterBase. In *Proceedings of the International Conference on Very Large Data Bases*, pages 347–355, Amsterdam, The Netherlands, August 1989.

[DEK90] W. Du, A. Elmagarmid, and W. Kim. Effects of Local Autonomy on Heterogeneous Distributed Database Systems. Technical Report ACT-OODS-EI-059-90, MCC, February 1990.

[DEK91] W. Du, A. Elmagarmid, and W. Kim. Maintaining Quasi Serializability in Multidatabase Systems. In *Proceedings of the 7th Intl. Conf. on Data Engineering*, pages 360–367, Kobe, Japan, April 1991.

[DELO89] W. Du, A. Elmagarmid, Y. Leu, and S. Ostermann. Effects of Autonomy on Global Concurrency Control in Heterogeneous Distributed Database Systems. In *Proceedings of the Second International Conference on Data and Knowledge Systems for Manufacturing and Engineering*, pages 113–120, Gaithersburg, Maryland, October 1989.

[EC92] A. Elmagarmid and J. Chen. The InterBase Parallel Language : Supporting the Flex Transaction Model and Beyond. Technical Report CSD-TR-92-017, Department of Computer Sciences, Purdue University, March 1992.

[ED91] A. Elmagarmid and W. Du. Integrity aspects of quasi serializability. *Information Processing Letters*, 38(1):23–28, April 1991.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase. In *Proceedings of the International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[GM83] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GMK88] H. Garcia-Molina and B. Kogan. Node Autonomy in Distributed Systems. In *Proceedings of the First International Symposium on Databases for Parallel and Distributed Systems*, pages 158–166, 1988.

[KLS90] H. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[LEB92] Y. Leu, A. Elmagarmid, and N. Boudriga. Specification and Execution of Transactions for Advanced Database Applications. *Information System*, March 1992.

[Lit85] W. Litwin. An Overview of the Multidatabase System MRDSM. In *Proceedings of the ACM Annual Conference*, pages 524–533, Denver, Colorado, 1985.

[LR82] T. Landers and R. Rosenberg. An Overview of Multibase. In H. Schneider, editor, *Distributed Data Systems*. North-Holland, 1982.

[PE91] R. Pezzoli and A. Elmagarmid. An InterBase Prototype of Bell Northern Research - A Case Study. Technical Report SERC-TR-95-P, Department of Computer Sciences, Purdue University, March 1991.

[ROEL90] M. Rusinkiewicz, S. Ostermann, A. Elmagarmid, and K. Loa. The Distributed Operation Language for Specifying Multi-System Applications. In *Proceedings of the First International Conference on System Integration*, April 1990.

[TLW87] M. Templeton, E. Lund, and P. Ward. Pragmatics of Access Control in Mermaid. *IEEE Data Engineering Bulletin*, 10(3):33–38, 1987.

# Appendix A : The Syntax of the IPL Language

The Backus-Naur Form (BNF) syntax of the IPL is as follows:

< *program* > ::= **program** < *type_defs* > < *subtrans_decls* >
< *depedency_decls* > < *final_status* > **endprogram**

< *type_defs* > ::= < *type_defs* > < *type_def* > | < *type_def* >

< *type_def* > ::= **class** < *user_type* > **of** < *type_list* > **endclass** ;

< *user_type* > ::= < *id* >

< *type_list* > ::= < *type_list* > < *a_type* > | < *a_type* >

< *a_type* > ::= < *var_list* > : < *basic_type* > ;

< *type* > ::= < *basic_type* > | < *user_type* > | < *compound_type* >

< *compound_type* > ::= **array of** < *basic_type* > | **array of** < *user_type* >

< *basic_type* > ::= **int** | **real** | **boolean** | **charString** | **bitString**

< *subtrans_decls* > ::= < *subtrans_decls* > < *subtrans_decl* > | < *subtrans_decl* >

< *subtrans_decl* > ::= **subtrans** < *id* > [ ( < *arg_list* > ) ] : < *type* >
**use** < *rsi* > **at** < *site* > < *subtrans_body* > **endsubtrans**

< *rsi* > ::= < *id* >

< *site* > ::= < *id* >

< *subtrans_body* > ::= [ < *time_constraint* > ] [ < *guard* > ]
< *executing* > [ < *confirm* > ] [ < *undo* > ]

< *executing* > ::= **beginexec** < *exec_body* > **endexec**

< *confirm* > ::= **beginconfirm** < *confirm_body* > **endconfirm**

< *undo* > ::= **beginundo** < *undo_body* > **endundo**

< *time_constraint* > ::= **before** < *time* >
| **between** < *time* > **to** < *time* >
| **after** < *time* >

< *time* > ::= " **timeofday** "

< *guard* > ::= **guard** < *ext_bool_expr* > ;

< *dependency_decls* > ::= **dependency** < *dependency_list* > **enddep**

< *dependency_list* > ::= < *dependency_list* > < *dependency_pair* >
| < *dependency_pair* >

< *dependency_pair* > ::= < *ext_bool_expr* > : < *id* > ;
| < *ext_bool_expr* > : **accept** ;

< *ext_bool_expr* > ::= < *ext_bool_expr* > **or** < *ext_bool_term* >
| < *ext_bool_term* >

< *ext_bool_term* > ::= < *ext_bool_term* > **and** < *ext_bool_factor* >
| < *ext_bool_factor* >

< *ext_bool_factor* > ::= < *operand* > < *compare_op* > < *operand* >
| ( < *ext_bool_expr* > )
| < *partial_succ_ext_bool_expr* >
| < *id* >
| **not** < *id* >

< *operand* > ::= < *value* > | [ < *id* > [ ( < *index* > ) ] . ] < *id* >

< *compare_op* > ::= > | >= | < | <= | <> | =

< *partial_succ_ext_bool_expr* > ::= ( < *number* > : < *subtrans_var_list* > )

< *subtrans_var_list* > ::= < *var_list* >

< *arg_list* > ::= < *var_list* >

< *var_list* > ::= < *var_list* > , < *id* > | < *id* >

< *final_status* > ::= **acceptable_sets** < *acceptable_sets* > **endaccs**

< *acceptable_sets* > ::= < *acceptable_sets* > , ( < *subtrans_list* > )
| ( < *subtrans_list* > )

< *subtrans_list* > ::= < *var_list* >

Keywords are in boldface, as **program**, **subtrans**. < *number* > and < *index* > can be any positive decimal number. < *value* > can be any real or integer number.

The lower bound of an array is 1; its upper bound will be determined automatically by the IPL program interpreter.

**timeofday** is in the format *mon day hh:mm:ss time-zone year*; as

Feb 27 11:31:46 GMT 1991. If an item is absent, then it takes the default value;

11:31:46 will be interpreted as

Feb 27 11:31:46 EST 1991 for a program executed in the eastern time zone.

# Appendix B : The Semantics of the IPL Language

We will explicate only those IPL syntactic constructs which are non-intuitive. We assume that the reader already has an understanding of context-free grammars.

- < *subtrans_decl* > ::= **subtrans** < *id* > [ ( < *arg_list* > ) ] : < *type* >
  **use** < *rsi* > **at** < *site* > < *subtrans_body* > **endsubtrans**

  This construct is used to define a subtransaction (e.g., $S_i$); its name, its type, its RSI server, and the RSI location are given by the < *id* >, < *type* >, < *rsi* >, and < *site* >, respectively. When $S_i$ is eligible to be scheduled, and both its time constraint and guard are evaluated *true*, DFTM initiates the process by obtaining permission from DCC. DFTM then opens a connection between the global transaction where $S_i$ is defined and an RSI service performed by the RSI server. Finally, using the RSI service as an intermediary, DFTM executes $S_i$ on the appropriate LSS.

  ( < *arg_list* > ), an option, defines the parameter list of $S_i$. As these parameters are actually the outputs of other subtransactions, the parameter list will consist of the names of these subtransactions, each of which must be unique.

  Each < *id* > has a double typed definition. Its explicit type is given by < *type* >, while its implicit type is an extended boolean type. That is, its value can be *true*, *false*, or *undefined*, respectively representing the success, failure, or undetermination of the < *executing* > term in the < *subtrans_body* > construct. When an < *id* > acts as an < *ext_bool_factor* > or appears in a < *subtrans_var_list* > construct, its implicit type is used; otherwise, it is defined by its explicit type. The use of the explicit type allows DFTM to process the output of the subtransaction as structured data rather than an uninterpreted string. Other subtransactions can then incorporate this output as a parameter in their execution. The implicit type of subtransaction, on the other hand, permits dependencies among subtransactions to be implemented as extended boolean expressions.

- < *subtrans_body* > ::= [ < *time_constraint* > ] [ < *guard* > ]
  < *executing* > [ < *confirm* > ] [ < *undo* > ]

  < *executing* > ::= **beginexec** < *exec_body* > **endexec**

  < *confirm* > ::= **beginconfirm** < *confirm_body* > **endconfirm**

  < *undo* > ::= **beginundo** < *undo_body* > **endundo**

  An < *exec_body* > construct constitutes a command text of the executing partial-subtransaction for a subtransaction $S_i$ in which the text is defined. The text will be executed when it is sent by the global transaction ($G_j$) to its RSI service.

A < *confirm_body* > construct, an option, constitutes a command text of the confirming partial-subtransaction for $S_i$. The text will be executed after $G_j$ decides to commit $S_i$ and sends the text to its RSI service.

An < *undo_body* > construct, also an option, constitutes a command text of the undoing partial-subtransaction for $S_i$. The text will be executed after $G_j$ decides to abort $S_i$ and sends the text to its RSI service.

The three partial-subtransactions will be discussed extensively in Section 5.4.

- < *ext_bool_expr* > ::= < *ext_bool_expr* > **or** < *ext_bool_term* >
  | < *ext_bool_term* >

  ...

  < *ext_bool_factor* > ::= < *operand* > < *compare_op* > < *operand* >

  ...

  This is an extended boolean expression definition. Each < *ext_bool_expr* >, < *ext_bool_term* >, or < *ext_bool_factor* > carries a value of *true*, *false*, or *undefined*. If an < *ext_bool_expr* > is evaluated as *undefined*, it will later be evaluated until its value is *true* or *false*.

  Both < *operand* > constructs in an < *ext_bool_factor* > should be of the same or compatible types. For example, if one is an integer and the other a real, the integer will be transformed to a real prior to comparison. If the two < *operand* >s are incompatible, then the value of the < *boolean_factor* > is *false*.

- < *time_constraint* > ::= **before** < *time* >
  | **between** < *time* > **to** < *time* >
  | **after** < *time* >

  < *guard* > ::= **guard** < *ext_bool_expr* > ;

  ...

  Both < *time_constraint* > and < *guard* > are options. When a subtransaction ($S_i$) is eligible to be scheduled, its time constraint and guard are evaluated. Any absent time constraint or guard is assigned the value *true*. If both options are *true*, then $S_i$ can be executed; if one of them is *false*, then a *false* value is bound to $S_i$, as if its execution had failed. If both are *undefined*, the execution of $S_i$ is delayed for later evaluation.

  < *time* > indicates the local time of the site where $S_i$ is executed. DFTM associates a time zone with each entry in the RSI Directory, enabling easy translation between the local and remote times.

  For the **before** construct, if the current time is before < *time* >, then < *time_constraint* > is *true*; otherwise, < *time_constraint* > is *false*.

  For the **between** construct, if the current time is before the first < *time* >, then < *time_constraint* > is *undefined*; if the current time is after the second < *time* >, then < *time_constraint* > is *false*; otherwise, < *time_constraint* > is *true*.

For the **after** construct, if the current time is before < *time* >, then < *time_constraint* > is *undefined*; otherwise, < *time_constraint* > is *true*.

Throughout the execution of $S_i$, DFTM continuously evaluates the < *time_constraint* >. If the evaluation returns *false* at some point, a timeout event occurs which signals the failure of the execution of $S_i$.

- < *dependency_pair* > ::= < *ext_bool_expr* > : < *id* > ;
  | < *ext_bool_expr* > : **accept** ;

  The construct defines an execution dependency. The subtransaction indicated by < *id* > is eligible to be scheduled if the < *ext_bool_expr* > on which it depends is *true*. If its governing < *ext_bool_expr* > is *false*, a *false* value is bound for the subtransaction, as if its execution has failed.

  Subtransactions can be executed in parallel if they do not depend on any < *ext_bool_expr* > or if the < *ext_bool_expr* > on which they depend are *true* and their time constraint and guard are both evaluated *true*.

  **accept**, a reserved word, indicates the final status of a global transaction ($G_j$). If its value is *true*, then $G_j$ succeeds; if *false*, then $G_j$ fails; otherwise, $G_j$ continues to run until **accept** becomes *true* or *false*. The **accept** is *true* or *false* if the < *ext_bool_expr* > on which it depends on is either *true* or *false*; otherwise, its value is *undefined*.

- < *partial_succ_ext_bool_expr* > ::= ( < *number* > : < *subtrans_var_list* > )

  The partially successful extended boolean expression is *true* only if at least < *number* > subtransactions in the < *subtrans_var_list* > construct have the value *true*. Its value is *false* if DFTM finds that there will not be < *number* > subtransactions in the < *subtrans_var_list* > construct that are *true*. Otherwise, its value is *undefined*.

- < *acceptable_sets* > ::= < *acceptable_sets* > , ( < *subtrans_list* > )
  | ( < *subtrans_list* > )

  When the **accept** is *undefined*, DFTM continues to execute subtransactions whenever possible until the **accept** becomes *true* or *false*. The *false* value indicates that the execution of the global transaction has failed, and thus all its subtransactions must be aborted. The *true* value indicates that the execution of the global transaction has succeeded; in this case, different acceptable sets, each consisting of a set of subtransactions, will be listed. The user is asked to determine the preferred set. The subtransactions in the preferred set will be committed; other subtransactions, of course, will be aborted.